Final Project – Progress Report 1

CS 385 – Computer Architecture

Daniel Kostecki, Thi Nguyen, and Sonia Leonato Soiras

**Tasks:**

Daniel Kostecki:

* Gate level conversions in the regfile (4x1 mux)
* Regfile implementation in project
* Report and diagrams

Thi Nguyen:

* 16-bit implementation of multiplexors using 4x1 mux
* Regfile D-flip-flops

Sonia Leonato Soiras:

* ALU conversions to 16-bit
* Gate level conversions in the ALU (4x1 mux and 2x1 mux)
* Truth table

**Descriptions:**

The instruction set architecture currently in place for our 16-bit machine allows for R-type instructions as well as the addi instruction. The list of R-type instructions that are currently supported and their respective opcode are and(0000), sub(0001), and(0010), or(0011), and slt(0111). The only I-format instruction that we have currently implemented is addi(0100).

R-type format: I-type format:

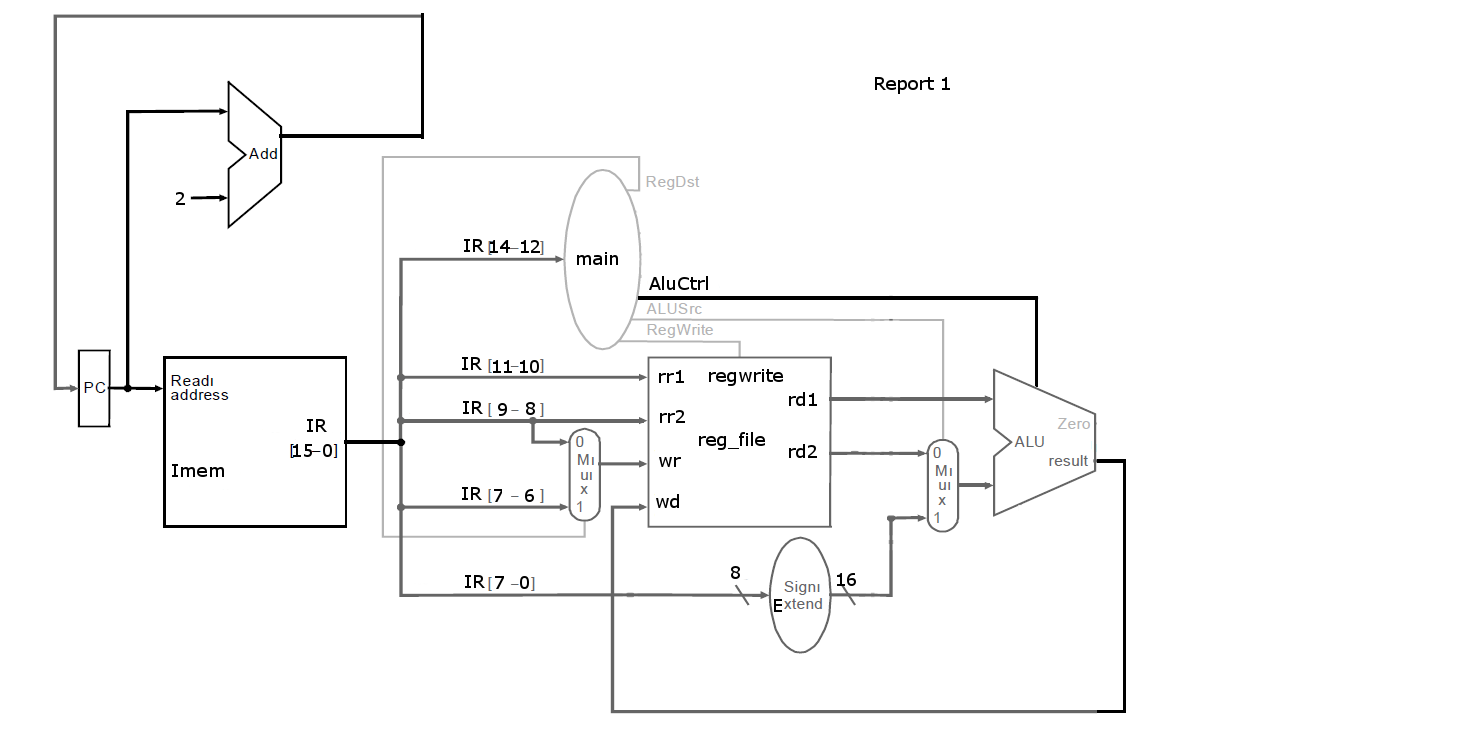
|  |  |  |  |
| --- | --- | --- | --- |
| Op | Rs | Rt | Address/Value |
| 4 | 2 | 2 | 8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Op | Rs | Rt | Rd | Unused |
| 4 | 2 | 2 | 2 | 6 |

The format of the instructions is dependent on the type if instruction. For example, in the r-type format we first get the op code (listed above). This is followed by the two registers which are being used. The final register is the destination. Because we have a 4-bit opcode followed by 3 2-bit registers this leaves us with 6-bits at the end of every r-type that is unused. For the i-type format this changes. The op code is still 4-bit, and there are still two registers (rs being used and rt being destination) but this is followed by an 8-bit value or address. For our current implementation, this will always be a value, as we have only implemented the addi instruction presently.

**Logic Diagrams/Truth Tables:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instr** | **RegDst** | **ALUSrc** | **MemReg** | **RegWrite** | **MemRead** | **MemWrite** | **Branch** | **ALUOp1** | **ALUOp2** |
| **Rtype** | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| **addi** | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |



**Verilog Source Code:**

/\*

By: Daniel Kostecki, Sonia Leonato, Thi Nguyen

Class: CS 385

Progress Report 1

\*/

//16-bit register

module reg\_file (rr1,rr2,wr,wd,regwrite,rd1,rd2,clock);

input [1:0] rr1,rr2,wr;

input [15:0] wd;

input regwrite,clock;

output [15:0] rd1,rd2;

wire [15:0] q1, q2, q3;

// registers

register r1 (wd,c1,q1),

r2 (wd,c2,q2),

r3 (wd,c3,q3);

// output port

mux4x1\_16bit mux1 (16'b0,q1,q2,q3,rr1,rd1),

mux2 (16'b0,q1,q2,q3,rr2,rd2);

// input port

decoder dec (wr[1],wr[0],w3,w2,w1,w0);

and a (regwrite\_and\_clock,regwrite,clock);

and a1 (c1,regwrite\_and\_clock,w1),

a2 (c2,regwrite\_and\_clock,w2),

a3 (c3,regwrite\_and\_clock,w3);

endmodule

// The register (16-bit D-flip flop)

module register(D,CLK,Q);

input [15:0]D;

input CLK;

output [15:0]Q;

D\_flip\_flop d1 (D[0],CLK,Q[0]),

d2 (D[1],CLK,Q[1]),

d3 (D[2],CLK,Q[2]),

d4 (D[3],CLK,Q[3]),

d5 (D[4],CLK,Q[4]),

d6 (D[5],CLK,Q[5]),

d7 (D[6],CLK,Q[6]),

d8 (D[7],CLK,Q[7]),

d9 (D[8],CLK,Q[8]),

d10(D[9],CLK,Q[9]),

d11(D[10],CLK,Q[10]),

d12(D[11],CLK,Q[11]),

d13(D[12],CLK,Q[12]),

d14(D[13],CLK,Q[13]),

d15(D[14],CLK,Q[14]),

d16(D[15],CLK,Q[15]);

endmodule

module D\_flip\_flop (D,CLK,Q);

input D,CLK;

output Q;

wire CLK1, Y;

not not1 (CLK1,CLK);

D\_latch D1(D,CLK, Y),

D2(Y,CLK1,Q);

endmodule

module D\_latch (D,C,Q);

input D,C;

output Q;

wire x,y,D1,Q1;

nand nand1 (x,D, C),

nand2 (y,D1,C),

nand3 (Q,x,Q1),

nand4 (Q1,y,Q);

not not1 (D1,D);

endmodule

// 16-bit ALU

module ALU (op,a,b,result,zero);

input [15:0] a;

input [15:0] b;

input [2:0] op;

output [15:0] result;

output zero;

wire c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15,c16;

ALU1 alu0 (a[0],b[0],op[2],op[1:0],set,op[2],c1,result[0]);

ALU1 alu1 (a[1],b[1],op[2],op[1:0],0,c1,c2,result[1]);

ALU1 alu2 (a[2],b[2],op[2],op[1:0],0,c2,c3,result[2]);

ALU1 alu3 (a[3],b[3],op[2],op[1:0],0,c3,c4,result[3]);

ALU1 alu4 (a[4],b[4],op[2],op[1:0],0,c4,c5,result[4]);

ALU1 alu5 (a[5],b[5],op[2],op[1:0],0,c5,c6,result[5]);

ALU1 alu6 (a[6],b[6],op[2],op[1:0],0,c6,c7,result[6]);

ALU1 alu7 (a[7],b[7],op[2],op[1:0],0,c7,c8,result[7]);

ALU1 alu8 (a[8],b[8],op[2],op[1:0],0,c8,c9,result[8]);

ALU1 alu9 (a[9],b[9],op[2],op[1:0],0,c9,c10,result[9]);

ALU1 alu10(a[10],b[10],op[2],op[1:0],0,c10,c11,result[10]);

ALU1 alu11(a[11],b[11],op[2],op[1:0],0,c11,c12,result[11]);

ALU1 alu12(a[12],b[12],op[2],op[1:0],0,c12,c13,result[12]);

ALU1 alu13(a[13],b[13],op[2],op[1:0],0,c13,c14,result[13]);

ALU1 alu14(a[14],b[14],op[2],op[1:0],0,c14,c15,result[14]);

ALUmsb alu15(a[15],b[15],op[2],op[1:0],0,c15,c16,result[15],set);

// If result = 0 then zero = 1

check\_zero cz(zero,result);

endmodule

// 1-bit ALU for bits 0-14

module ALU1 (a,b,binvert,op,less,carryin,carryout,result);

input a,b,less,carryin,binvert;

input [1:0] op;

output carryout,result;

wire sum, a\_and\_b, a\_or\_b, b\_inv;

not not1(b\_inv, b);

mux2x1 mux1(b,b\_inv,binvert,b1);

and and1(a\_and\_b, a, b);

or or1(a\_or\_b, a, b);

fulladder adder1(sum,carryout,a,b1,carryin);

mux4x1 mux2(a\_and\_b,a\_or\_b,sum,less,op[1:0],result);

endmodule

// 1-bit ALU for the most significant bit

module ALUmsb (a,b,binvert,op,less,carryin,carryout,result,sum);

input a,b,less,carryin,binvert;

input [1:0] op;

output carryout,result,sum;

wire sum, a\_and\_b, a\_or\_b, b\_inv;

not not1(b\_inv, b);

mux2x1 mux1(b,b\_inv,binvert,b1);

and and1(a\_and\_b, a, b);

or or1(a\_or\_b, a, b);

fulladder adder1(sum,carryout,a,b1,carryin);

mux4x1 mux2(a\_and\_b,a\_or\_b,sum,less,op[1:0],result);

endmodule

module check\_zero (zero,result);

input [15:0] result;

wire z[14:0];

output zero;

or g5(z[0],result[0],result[1]);

or g6(z[1],result[2],result[3]);

or g7(z[2],result[4],result[5]);

or g8(z[3],result[6],result[7]);

or g9(z[4],result[8],result[9]);

or g1(z[5],result[10],result[11]);

or g2(z[6],result[12],result[13]);

or g3(z[7],result[14],result[15]);

or g4(z[8],z[0],z[1]);

or g10(z[9],z[2],z[3]);

or g11(z[10],z[4],z[5]);

or g12(z[11],z[6],z[7]);

or g13(z[12],z[8],z[9]);

or g14(z[13],z[10],z[11]);

or g15(z[14],z[12],z[13]);

xnor g8(zero,z[14],0);

endmodule

// Arithmetic

module halfadder (S,C,x,y);

input x,y;

output S,C;

xor (S,x,y);

and (C,x,y);

endmodule

module fulladder (S,C,x,y,z);

input x,y,z;

output S,C;

wire S1,D1,D2;

halfadder HA1 (S1,D1,x,y),

HA2 (S,D2,S1,z);

or g1(C,D2,D1);

endmodule

// Multiplexers

module mux4x1 (a\_and\_b, a\_or\_b,sum, less, op[1:0], result);

input a\_and\_b,a\_or\_b,sum,less;

input [1:0] op;

wire a,b,c,d,notOp0,notOp1;

output result;

not g1(notOp0,op[0]); // S0

not g2(notOp1,op[1]); // S1

and g3(a,a\_and\_b,notOp1,notOp0), // AND operation - 00

g4(b,a\_or\_b,notOp1,op[0]), // OR operation - 01

g5(c,sum,op[1],notOp0), // ADD operation - 10

g6(d,less,op[1],op[0]); // SLT operation - 11

or g7(result,a,b,c,d);

endmodule

module mux4x1\_16bit (i0,i1,i2,i3,select,y);

input [15:0] i0,i1,i2,i3;

input [1:0] select;

output [15:0] y;

mux4x1 m1(i0[0], i1[0], i2[0], i3[0], select,y[0]);

mux4x1 m2(i0[1], i1[1], i2[1], i3[1], select,y[1]);

mux4x1 m3(i0[2], i1[2], i2[2], i3[2], select,y[2]);

mux4x1 m4(i0[3], i1[3], i2[3], i3[3], select,y[3]);

mux4x1 m5(i0[4], i1[4], i2[4], i3[4], select,y[4]);

mux4x1 m6(i0[5], i1[5], i2[5], i3[5], select,y[5]);

mux4x1 m7(i0[6], i1[6], i2[6], i3[6], select,y[6]);

mux4x1 m8(i0[7], i1[7], i2[7], i3[7], select,y[7]);

mux4x1 m9(i0[8], i1[8], i2[8], i3[8], select,y[8]);

mux4x1 m10(i0[9], i1[9], i2[9], i3[9], select,y[9]);

mux4x1 m11(i0[10],i1[10],i2[10],i3[10], select,y[10]);

mux4x1 m12(i0[11],i1[11],i2[11],i3[11],select,y[11]);

mux4x1 m13(i0[12],i1[12],i2[12],i3[12],select,y[12]);

mux4x1 m14(i0[13],i1[13],i2[13],i3[13],select,y[13]);

mux4x1 m15(i0[14],i1[14],i2[14],i3[14],select,y[14]);

mux4x1 m16(i0[15],i1[15],i2[15],i3[15],select,y[15]);

endmodule

module mux2x1 (A,B,select,OUT);

input A,B,select;

output OUT;

wire notS,andA,andB;

not g1(notS,select);

and g2(andA,A,notS),

g3(andB,B,select);

or g4(OUT,andA,andB);

endmodule

module mux2x1\_2bit (A,B,select,y);

input [1:0] A,B;

input select;

output [1:0] y;

mux2x1 mux1(A[0], B[0], select, y[0]),

mux2(A[1], B[1], select, y[1]);

endmodule

module mux2x1\_16bit (A, B, select, y);

input [15:0] A,B;

input select;

output [15:0] y;

mux2x1 mux1(A[0], B[0], select, y[0]),

mux2(A[1], B[1], select, y[1]),

mux3(A[2], B[2], select, y[2]),

mux4(A[3], B[3], select, y[3]),

mux5(A[4], B[4], select, y[4]),

mux6(A[5], B[5], select, y[5]),

mux7(A[6], B[6], select, y[6]),

mux8(A[7], B[7], select, y[7]),

mux9(A[8], B[8], select, y[8]),

mux10(A[9], B[9], select, y[9]),

mux11(A[10], B[10], select, y[10]),

mux12(A[11], B[11], select, y[11]),

mux13(A[12], B[12], select, y[12]),

mux14(A[13], B[13], select, y[13]),

mux15(A[14], B[14], select, y[14]),

mux16(A[15], B[15], select, y[15]);

endmodule

module decoder (S1,S0,D3,D2,D1,D0);

input S0,S1;

output D0,D1,D2,D3;

not n1 (notS0,S0),

n2 (notS1,S1);

and a0 (D0,notS1,notS0),

a1 (D1,notS1, S0),

a2 (D2, S1,notS0),

a3 (D3, S1, S0);

endmodule

//Main Control Unit

module MainControl (op,control);

input [2:0] op;

output reg [5:0] control;

//Control is in the format of: RegDst, AluSrc, RegWrite, AluCtrl

always @(op) case (op)

//R-types

3'b010: control <= 6'b101010; // add

3'b110: control <= 6'b101110; // sub

3'b000: control <= 6'b101000; // and

3'b001: control <= 6'b101001; // or

3'b111: control <= 6'b101111; // slt

//I-type

3'b100: control <= 6'b011010; // addi

endcase

endmodule

module CPU (clk, AluOut, IR);

input clk;

output [15:0] AluOut, IR;

reg[15:0] PC;

reg[15:0] Imem[0:511];

wire [15:0] IR, NextPC, A, B, AluOut, RD2, SignExtend;

wire [2:0] AluCtrl;

wire [1:0] WR;

//To test

initial begin

Imem[0] = 16'b0100000100001111; // addi $t1, $0, 15 # $t1=15

Imem[1] = 16'b0100001000000111; // addi $t2, $0, 7 # $t2= 7

Imem[2] = 16'b0000011011000000; // and $t3, $t1, $t2 # $t3= 7

Imem[3] = 16'b0110011110000000; // sub $t2, $t1, $t3 # $t2= 8

Imem[4] = 16'b0001101110000000; // or $t2, $t2, $t3 # $t2=15

Imem[5] = 16'b0010101111000000; // add $t3, $t2, $t3 # $t3=22

Imem[6] = 16'b0111111001000000; // slt $t1, $t3, $t2 # $t1= 0

Imem[7] = 16'b0111101101000000; // slt $t1, $t2, $t3 # $t1= 1

end

initial PC = 0;

assign IR = Imem[PC>>1];

mux2x1\_2bit muxWR (IR[9:8], IR[7:6], RegDst, WR);

mux2x1\_16bit muxB (RD2, SignExtend, AluSrc, B);

assign SignExtend = {{8{IR[7]}},IR[7:0]};

reg\_file rf (IR[11:10], IR[9:8], WR, AluOut, RegWrite, A, RD2, clk);

ALU fetch (3'b010, PC, 16'b10, NextPC, Unused);

ALU exec (AluCtrl, A, B, AluOut, Zero);

MainControl main (IR[14:12], {RegDst, AluSrc, RegWrite, AluCtrl});

// Program counter

always @(negedge clk) begin

PC <= NextPC;

end

endmodule

// To test

module testing ();

reg clock;

wire [15:0] WD,IR;

CPU test\_cpu(clock,WD,IR);

always #1 clock = ~clock;

initial begin

$display ("Time Clock IR WD");

$monitor ("%2d %b %h %h", $time,clock,IR,WD);

clock = 1;

#14 $finish;

end

endmodule

/\* Test results

Time Clock IR WD

0 1 410f 000f

1 0 4207 0007

2 1 4207 0007

3 0 06c0 0007

4 1 06c0 0007

5 0 6780 0008

6 1 6780 0008

7 0 1b80 000f

8 1 1b80 000f

9 0 2bc0 0016

10 1 2bc0 0016

11 0 7e40 0000

12 1 7e40 0000

13 0 7b40 0001

14 1 7b40 0001

\*/

**Test Results:**